

Numpy z TŁUMACZENIEM



NumPy - podstawowe narzędzie ANALITYKA DANYCH, INŻYNIERA UCZENIA MASZYNOWEGO. To biblioteka obliczeń numerycznych Pythona.

- Na stronie numpy.org są informacje o niej. W narzędziu COLAB są wszystkie biblioteki, o które będziemy przerabiać, czyli mogę z nich korzystać bez instalacji
- Mogę zainstalować też lokalnie używając np. `pip install numpy`

Poniżej:

- importuję bibliotekę i daję alias
- sprawdzam wersję biblioteki
- SHIFT + ENTER to uruchomienie pojedynczych komórek

```
[ ] import numpy as np  
    np.__version__
```

```
'1.23.5'
```

Sprawdzam co jest dostępne w NumPy.

- Jeśli nie dam print() wszystko wyświetli mi się linia pod linią

```
[ ] print(dir(np))
```

```
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL', 'ERR
```

✓ I. OPERACJE NA TABLICACH

- 1. Array JEDNOWYMIAROWE

Funkcja array() - to funkcja do tworzenia tablic

- help(np.array) - pomoc do funkcji array() do tego jak taką tablicę utworzyć, przykłady jakie parametry podać
- nie dodałem ja aby nie obciążać GitHuba

TEORIA:

- JEDNORODNOŚĆ w tablicy - każdy element musi być tego samego typu
- Wymiary w bibliotece NumPy nazywa się osiami (axes)
- Takie obiekty należą do klasy zwanej handy array

PONIŻEJ:

- tablica tj. lista - podana w funkcji do tworzenia tablic array() - o jednym wymiarze przypisana do zmiennej x

- tablica jednowymiarowa tj. lista bez zagnieżdżeń

```
[ ] x = np.array([1, 3])  
x
```

```
array([1, 3])
```

```
[ ] print(x)
```

```
[1 3]
```

Funkcja `type()` - sprawdza typ obiektu

- Klasa `ndarray`. Ma ona wiele atrybutów poniżej wyświetlam

```
[ ] type(x)
```

```
numpy.ndarray
```

Metoda `ndim` - wyświetla wymiar tablicy, choćby tablicy `x`

```
[ ] x.ndim
```

```
1
```

`shape` - pozwala wyświetlić kształt jako tuple (). Mam 2 - bo mam dwa elementy

```
[ ] x.shape
```

```
(2,)
```

`size` - wyświetla liczbę elementów naszej tablicy

```
[ ] x.size
```

```
2
```

`dtype` - wyświetla typ danych tablicy

```
[ ] x.dtype
```

```
dtype('int64')
```

2 Array Dwuwymiarowe

Jest w tych tablicach domyślna implementacja macierzy

- lista w liście
- przydatne są do tworzenia sieci neuronowych, które składają się z wielu operacji mnożenia i dodawania macierzy

Ponizej:

- tablica dwuwymiarowa o dwóch WIERSZACH i dwóch KOLUMNACH

```
[ ] np.array([[1, 2], [-3, 1]])
```

```
array([[ 1,  2],  
       [-3,  1]])
```

```
[ ] x = np.array([[1, 2], [-3, 1]])  
x
```

```
array([[ 1,  2],  
       [-3,  1]])
```

- ndim - wyświetlenie wymiarów tablicy dwuwymiarowej x.
- shape - wyświetlenie kształtu danych: dwa WIERSZE i dwie KOLUMNY

```
[ ] x.ndim
```

2

```
[ ] x.shape
```

```
(2, 2)
```

Tworzę tablicę DWUWYMIAROWĄ, która ma: trzy KOLUMNY i dwa WIERSZE

- shape - kształt moich danych to dwa WIERSZE i trzy KOLUMNY

```
[ ] x = np.array([[1, 2, 3], [4, 2, 1]])  
print(x)
```

```
[[1 2 3]  
 [4 2 1]]
```

```
[ ] x.shape
```

```
(2, 3)
```

3. TABLICE TRÓJWYMIAROWE

- pozwalają one przechowywać takie dane jak np. OBRAZ

```
[ ] x = np.array(  
    [[ [4, 3, 1],  
        [3, 1, 2]],  
    [[ [4, 1, 3],  
        [4, 2, 1]]]  
)  
x
```

```
array([[[4, 3, 1],  
        [3, 1, 2]],  
       [[4, 1, 3],  
        [4, 2, 1]]])
```

```
[[4, 1, 3],  
 [4, 2, 1]])
```

ndim - wyświetlam atrybuty tablicy, mam 3 bo jest to tablica TRÓJWYMIAROWA

```
[ ] x.ndim
```

3

```
[ ] x = np.array(  
    [[4, 3, 1],  
     [3, 1, 2]],  
    [[4, 1, 3],  
     [4, 2, 1]],  
    [[3, 3, 1],  
     [4, 3, 2]])  
x
```

```
array([[4, 3, 1],  
       [3, 1, 2]],  
       [[4, 1, 3],  
       [4, 2, 1]],  
       [[3, 3, 1],  
       [4, 3, 2]])
```

```
[ ] x.ndim
```

3

```
[ ] x.ndim
```

3

shape - kształt tablicy: 3 ELEMENTY, 2 WIERSTY, 3 KOLUMNY

```
[ ] x.shape
```

(3, 2, 3)

▼ [Bez tytułu] Y DANYCH

Tworzę tablicę:

- liczb CAŁKOWITYCH
- składającą się z float

```
[ ] A = np.array([1, 2, 3])  
A.dtype
```

```
dtype('int64')
```

```
[ ] A = np.array([1.0, 2.3, 3.3])  
A.dtype
```

```
dtype('float64')
```

Ustawiam dodatkowy parametr dtype, przekazując go do funkcji array()

- nie muszę tu przekazywać danych jako float

```
[ ] A = np.array([1, 2, 3], dtype='float')  
A.dtype
```

```
dtype('float64')
```

Poniżej: wyświetlam tablicę

```
[ ] A
```

```
array([1., 2., 3.])
```

Konwertuję dane typ LICZB ZŁOŻONYCH:

- Sprawdź czym jest ten typ?

```
[ ] A = np.array([1, 2, 3], dtype='complex')  
A.dtype
```

```
dtype('complex128')
```

Konwertuję ustawione dane z typu FLOAT na typ INT

- Przy wyświetleniu tablicy A widzę same int'y

```
[ ] A = np.array([1.0, 2.3, 3.3], dtype='int')  
A.dtype
```

```
dtype('int64')
```

```
[ ] A
```

```
array([1, 2, 3])
```

Tworzę tablicę z typem BOOL

```
[ ] A = np.array([True, False])  
A.dtype
```

```
dtype('bool')
```

Używam wbudowanego typu z biblioteki np czyli int8

- ten typ jest przydatny przy tworzeniu tablicy, która opisuje zdjęcia, i kiedy chcę przedstawić konkretne wartości w pixelach i nie zająć za dużo miejsca - można pobrać 8-bitową liczbę całkowitą

```
[ ] A = np.array([24, 120, 230], dtype=np.int8)  
A.dtype
```

```
dtype('int8')
```

Częściej używany typ uint8 - czyli użycie wartości od 0 do 255 (przydatne do opisu obrazów)

```
[ ] A = np.array([24, 120, 230], dtype=np.uint8)  
A.dtype
```

```
dtype('uint8')
```


✓ III TWORZENIE TABLIC NumPy ndarray

- POWYŻEJ: podałem przykłady ręcznego tworzenia tablic przy pomocy funkcji `array()`
- PONIŻEJ: przykłady funkcji wbudowanych pozwalających automatyzować ten proces tworzenia tablic

Funkcja `zeros()` - pozwala utworzyć tablicę składającą się z dowolnego rozmiaru samych ZER

- ROZMIAR: 4 wiersze, 10 kolumn
- Domyślnym typem danych jest tu `FLOAT`
- Nie trzeba ręcznie implementować ZER

```
[ ] np.zeros(shape=(4, 10))
```

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Poniżej:

- ustawiam wyświetlenie zer jako liczby CAŁKOWITE
- funkcja ta służy do tego, jeśli chcę utworzyć ZDJĘCIE 600px na 800px, które zawiera CZARNE TŁO to daję numpy zeros()

```
[ ] #@title
np.zeros(shape=(4, 10), dtype="int")

array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Funkcja ones(shape=())

- działa podobnie jak powyższa ale wstawiam same JEDYNKI
- mam tablicę 5 na 5 - macierz kwadratową
- pierwszy przykład z FLOAT'ami kolejny z INT'ami

```
[ ] np.ones(shape=(5, 5))

array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
[ ] np.ones(shape=(5, 5), dtype='int')
```

```
array([[1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1]])
```

Funkcja full(shape=())

- jest bardziej uniwersalna od powyższych
- pozwala wypełnić tablicę konkretną wartością
- ODP - mam tablicę rozmiaru 3 na 3 wypełnioną wartością 4

[Bez tytułu]

```
[ ] np.full(shape=(3, 3), fill_value=4, dtype='int')
```

```
array([[4, 4, 4],  
       [4, 4, 4],  
       [4, 4, 4]])
```

Funkcja arrange()

- pozwala generować dane w postaci tablicy numpy array
- jeśli podaję 11 to generuje dane od 0 do 11

```
[ ] np.arange(11)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

- Zmieniam element na 6 od którego zaczynam generowania liczb
- Dodaję parametr kroku

```
[ ] np.arange(start=6, stop=11)
```

```
array([ 6,  7,  8,  9, 10])
```

```
[ ] np.arange(start=10, stop=100, step=10)
```

```
array([10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Ustawienie KROKU JAKO WARTOSCI UJEMNEJ - zaczynam od wartości wyższej

```
[ ] np.arange(start=100, stop=10, step=-10)
```

```
array([100,  90,  80,  70,  60,  50,  40,  30,  20])
```

Ustawienie kroku jako float

```
[ ] np.arange(start=0, stop=1, step=0.05)
```

```
array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,  
       0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95])
```

Funkcja linspace()

- wskazuję w parametrze ile elementów chcę wygenerować z określonego przedziału

```
[ ] np.linspace(start=0, stop=1, num=10)
```

```
array([0.          , 0.11111111, 0.22222222, 0.33333333, 0.44444444,  
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.          ])
```

```
[ ] np.linspace(start=0, stop=1, num=11)
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Generowanie i wyświetlenie 15 elementów od 0 do 15

```
[ ] A = np.arange(15)
```

```
A
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Metoda reshape()

- zmiana rozmiaru kształtu na: 3 na , tak by iloczyn dawał 15
- do nawiasów przekazany został rozmiar jako tupla

```
[ ] A.reshape((3, 5))
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14]])
```

Chcę mieć 3 wiersze i nich Python generuje odpowiednia liczbę drugiego parametru

```
[ ] A.reshape((3, -1))
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14]])
```

Nie chcę obliczać pierwszego parametru samemu stąd daję -1. Podaję 3 kolumny jako drugi parametr

```
[ ] #@title  
A.reshape((-1, 3))
```

```
array([[ 0,  1,  2],  
       [ 3,  4,  5],  
       [ 6,  7,  8],  
       [ 9, 10, 11],  
       [12, 13, 14]])
```

✓ IV. PODSTAWOWE OPERACJE NA TABLICACH

Tworzę iwypisuję dwie tablice

```
[ ] #@title
A = np.array([3, 1, 4, 2])
B = np.array([3, -1, 3, 3])
print(A)
print(B)
```

```
[3 1 4 2]
[ 3 -1  3  3]
```

OPERACJE ARYTMETYCZNE:

Przy operacji choćby: dodawaniai należy pamiętać aby ROZMIAR DANYCH się zgadzał

```
[ ] A + B
```

```
array([6, 0, 7, 5])
```

```
→ array([6, 0, 7, 5])
```

```
[ ] A - B
```

```
array([ 0,  2,  1, -1])
```

```
[ ] A * B
```

```
array([ 9, -1, 12,  6])
```

```
[ ] A / B
```

```
array([ 1.          , -1.          ,  1.33333333,  0.66666667])
```

add()

- metoda dodawania dwóch tablic
- Kolejne metody to: subtract(); divide(); multiplay() tj. odejmowania, dzielenia, dwóch tablic

```
[ ] np.add(A, B)
```

```
array([6, 0, 7, 5])
```

```
[ ] np.subtract(A, B)
```

```
array([ 0,  2,  1, -1])
```

```
[ ] np.divide(A, B)
```

```
array([ 1.          , -1.          ,  1.33333333,  0.66666667])
```

Ponizej:

- do całej tablicy dodeję pewną stałą tj. 3
- również każdy element mogę pomnożyć przez jakąś liczbę

```
[ ] A + 3
```

```
array([6, 4, 7, 5])
```

```
[ ] 2 * A
```

```
array([6, 2, 8, 4])
```

```
[ ] A + 3*B
```

```
array([12, -2, 13, 11])
```

Buduję dwie tablice - macierze kwadratowe o tej samej liczbie wierszy i tej samej liczbie kolumn

```
[ ] X = np.array([[1, 3], [-2, 0]])  
Y = np.array([[6, 0], [-1, 2]])  
print(X, '\n')  
print(Y)
```

```
[[ 1  3]  
 [-2  0]]
```

```
[[ 6  0]  
 [-1  2]]
```

Mnożenie elementu po elemencie

```
[ ] X * Y
```

```
array([[6, 0],  
       [2, 0]])
```


Metoda dot()

- mnożenie macierzowe - mnożeniei wiersz razy kolumna i wiersz razy kolumna

```
[ ] np.dot(X, Y)
```

```
array([[ 3,  6],  
       [-12,  0]])
```

Użycie na konkretnej macierzy metody dot()

- nie jest to to samo co powyżej. MNożenie macierzy nie ejst przemienne

```
[ ] X.dot(Y)
```

```
array([[ 3,  6],  
       [-12,  0]])
```

```
[ ] Y.dot(X)
```

```
array([[ 6, 18],  
       [-5, -3]])
```

Symbol @ również pozwala pomnożyć macierze

```
[ ] X @ Y
```

```
array([[ 3,  6],  
       [-12,  0]])
```

✓ V. GENEROWANIE LICZB PESUDOLOSOWYCH

Ustawiam konkretne ziarno losowania aby za każdym razem : uzyskać ten sam wynik

```
[ ] np.random.seed(0)
```

Moduł random pozwala generować pseudolosowe liczby

```
[ ] np.random.randn()
```

```
1.764052345967664
```

Funkcja randin() generuje liczby z rozkładu normalnego

- Generuję 10 liczb od 0 do 1

```
[ ] np.random.randn(10)
```

```
array([ 0.40015721,  0.97873798,  2.2408932 ,  1.86755799, -0.97727788,  
        0.95008842, -0.15135721, -0.10321885,  0.4105985 ,  0.14404357])
```

Generowanie pseudolosowych liczb: 10 WIERSZY i 4 KOLUMNY

```
[ ] np.random.rand(10, 4)
```

```
array([[0.56804456, 0.92559664, 0.07103606, 0.0871293 ],
       [0.0202184 , 0.83261985, 0.77815675, 0.87001215],
       [0.97861834, 0.79915856, 0.46147936, 0.78052918],
       [0.11827443, 0.63992102, 0.14335329, 0.94466892],
       [0.52184832, 0.41466194, 0.26455561, 0.77423369],
       [0.45615033, 0.56843395, 0.0187898 , 0.6176355 ],
       [0.61209572, 0.616934 , 0.94374808, 0.6818203 ],
       [0.3595079 , 0.43703195, 0.6976312 , 0.06022547],
       [0.66676672, 0.67063787, 0.21038256, 0.1289263 ],
       [0.31542835, 0.36371077, 0.57019677, 0.43860151]])
```

```
[ ] np.random.rand()
```

```
0.9883738380592262
```

```
[ ] np.random.rand(10)
```

```
array([0.10204481, 0.20887676, 0.16130952, 0.65310833, 0.2532916 ,
       0.46631077, 0.24442559, 0.15896958, 0.11037514, 0.65632959])
```

Przekazuję drugi element i otrzymuje tablicę dwuwymiarową

```
[ ] np.random.rand(10, 2)
```

```
array([[0.13818295, 0.19658236],
       [0.36872517, 0.82099323],
       [0.09710128, 0.83794491],
       [0.09609841, 0.97645947],
       [0.4686512 , 0.97676109],
       [0.60484552, 0.73926358],
       [0.03918779, 0.28280696],
       [0.12019656, 0.2961402 ],
       [0.11872772, 0.31798318],
       [0.41426299, 0.0641475 ]])
```

[Bez tytułu]

Jeśli dam 10 to zwraca liczbę całkowitą od 0 do 10 mniejszą niż 10

```
[ ] np.random.randint(10)
```

```
2
```

- low - parametr początkowy
- high - parametr końcowy

```
[ ] np.random.randint(low=10, high=101)
```

```
56
```

Parametr size - określa rozmiar generowanych danych

```
[ ] np.random.randint(low=10, high=102, size=8)

array([30, 91, 60, 37, 24, 51, 68, 75])
```

Metoda choice() - wybiera losowo element z obiektu, który przekażę:

- wartości liczbowe;
- wartości tekstowe

```
[ ] np.random.choice([5, 10, 14, 3])

5
```

```
[ ] np.random.choice(['python', 'java', 'sql', 'C#'])

'sql'
```

Funkcja arrange()

```
[ ] data = np.arange(10)
data

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Funkcja shuffle()

- jako parametr podaję obiekt, który chce przetasować
- należy tu być ostrożnym bo zmiana zachodzi na obiekcie

```
[ ] np.random.shuffle(data)
```

```
[ ] data

array([1, 5, 9, 4, 0, 7, 2, 3, 8, 6])
```

✓ VI. FUNKCJE W BIBLOTECE NUMPY

Funkcja exp()

- funkcja wykorzystywana w ML i DL
- Jeśli dam 1 to funkcja zwraca STAŁA EULERA

```
[ ] np.exp(1)

2.718281828459045
```

Funkcja `sql()`

- zwraca float; pierwiastek kwadratowy

```
[ ] np.sqrt(9)
```

```
3.0
```

Funkcja `all()`

- jest sposobem testowania danych
- zwraca wartość logiczną. Funkcja ta przechodzi po obiekcie i bada każdy element i zwraca wartość logiczną (true lub false). Jeśli każdy element zwróci true to zostanie zwrócona prawda, jeśli jakiś element zwróci false to zostanie zwrócony false
- Sprawdź w dokumentacji i w internecie jak dokładnie funkcjonuje owa funkcja!

```
[ ] np.all([2, 3, 1])
```

```
True
```

Funkcja `any()`

- sprawdza czy jakikolwiek obiekt jest prawdą

```
[ ] np.any([1, 3, 4])
```

```
True
```

```
[ ] np.any([0, 0, 0])
```

```
False
```

Generuję obiekt w przedziale od 0 do 1 z 5 elementami

- może to być wynik prawdopodobieństwa otrzymanai klasy dla danej próbki

```
[ ] A = np.random.rand(5)  
A
```

```
array([0.26473016, 0.39782075, 0.55282148, 0.16494046, 0.36980809])
```

Funkcja argmax()

- Podciąga indx, gdzie tablica otrzymuje wartość najwyższą
- ZADANIE: odkryj logikę działania poniższych z max i min funkcji!

```
[ ] np.argmax(A)
```

```
2
```

Wyciągam tę wartość z tablicy, z obiektu A

```
[ ] A[np.argmax(A)]
```

```
0.5528214798875715
```

```
[ ] np.argmin(A)
```

```
3
```

Funkcja argsort()

- pozwala sortować elementy podając odpowiedni index na przykład index 3

```
[ ] np.argsort(A)
```

```
array([3, 0, 4, 1, 2])
```

Ponizej funkcje zwracające z tablicy WARTOŚĆ:

- NAJWIĘKSZĄ; NAJMNIEJSZĄ; ŚREDNIĄ; MEDIANĘ; ODCHYLENIE STANDARDOWE

```
[ ] np.max(A)
```

```
0.5528214798875715
```

```
[ ] np.min(A)
```

```
0.16494046024188413
```

```
[ ] np.mean(A)
```

```
0.35002418998397483
```

```
[ ] np.median(A)
```

```
0.36980809274834003
```

```
[ ] np.std(A)
```

```
0.13063974345454746
```

✓ VII. INDEKSOWANIE I WYCINANIE TABLIC

Metoda arrange()

- generuję do pracy dane do pracy - tablica jednowymiarowa o 20 elementach

```
[ ] A = np.arange(20)
A

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Ponizej kolejno:

- Wycinam drugi element
- Wycinam od drugiego elementu
- Wycinam do drugiego elementu
- Wycinam tylko index 0 i index 2
- Wycinam ostatni element za pomocą indexowania od końca tj. -1
- Wycinam od 11 do 15 gdzie ostatni element nie jest wliczany

```
[ ] A[2]
```

```
2
```

```
[ ] A[2:]
```

```
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
       19])
```

```
A[:2]
```

```
array([0, 1])
```

```
[ ] A[[0, 2]]
```

```
array([0, 2])
```

```
[ ] A[-1]
```

```
19
```

```
[ ] A[11:15]
```

```
array([11, 12, 13, 14])
```

Metoda reshape()

- tworzę o wymiarach 4 na 5

```
[ ] A = A.reshape(4, 5)
A
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```


Wycinam pierwszy wiersz

```
[ ] A[0]
```

```
array([0, 1, 2, 3, 4])
```

Wycinam drugi wiersz

```
[ ] A[1]
```

```
array([5, 6, 7, 8, 9])
```

Wycinam pierwszą kolumnę. Podaję: pierwszy parametr dwukropek i index kolumny

```
[ ] A[:,0]
```

```
array([ 0,  5, 10, 15])
```

Wycinam ostatnią kolumnę

```
[ ] A[:, -1]
```

```
array([ 4,  9, 14, 19])
```

Wycinam przedostatnią kolumnę

```
[ ] A[:, -2]
```

Wycinam konkretny element: podaję wiersz z index'em i kolumnę z indexem

```
[ ] A[1, 1]
```

6

```
[ ] A[1, 3]
```

8

Podaję przedziały: od index'u wiersza 1:3 i index kolumny 1:4

```
[ ] A[1:3, 1:4]
```

```
array([[ 6,  7,  8],  
       [11, 12, 13]])
```

Do tych wartości mogę przypisać inne rzeczy:

- wycinam 7 tj. podaję numer wiersza tj.1 i numer kolumny tj. 2. Przypisuję do niej 14
- pod spodem daję A i wyświetlam całą tablicę

```
[ ] A[1, 2] = 14  
A
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6, 14,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

✓ VIII. Iteracja po tablicach, zmiana wielkości oraz maski logiczne

A. ITERACJA PO TABLICACH

```
[ ] A[1, 2] = 14  
A
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6, 14,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

Iteruję wiersz po wierszu

```
[ ] for row in A:  
    | | print(row)
```

```
[0 1 2 3 4]  
[ 5  6 14  8  9]  
[10 11 12 13 14]  
[15 16 17 18 19]
```

Iteruję po zerowych index'ach, elementach wiersza

```
[ ] for row in A:  
    | | print(row[0])
```

```
0  
5  
10  
15
```

Wyświetlam trzy pierwsze elementy

```
[ ] for row in A:  
    | | print(row[:3])
```

```
[0 1 2]  
[ 5  6 14]  
[10 11 12]  
[15 16 17]
```

Iteruję po każdym elemencie

- Dodaję atrybut flat - sprawdź co ten atrybut oznacza!

```
[ ] for item in A.flat:  
    | | print(item)
```

0
1
2
3
4
5
6
14
8
9
10
11
12
13
14
15
16
17
18
19

B. ZMIANA ROZMIARU TABLIC

```
[ ] A
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6, 14,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

```
[ ] A.shape
```

(4, 5)

Używam metody do zmiany rozmiaru tablicy

```
[ ] A.reshape(5, 4)
```

```
array([[ 0,  1,  2,  3],  
       [ 4,  5,  6, 14],  
       [ 8,  9, 10, 11],  
       [12, 13, 14, 15],  
       [16, 17, 18, 19]])
```

Metoda ravel()

- metoda pomagająca wypłaszczyć dane do postaci jednowymiarowej
- to metoda odwrotna do metody reshape()

```
[ ] A.ravel()
```

```
array([ 0,  1,  2,  3,  4,  5,  6, 14,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
       17, 18, 19])
```

Zamiana wierszy na kolumny i kolumn na wiersze

```
[ ] A.T
```

```
array([[ 0,  5, 10, 15],  
       [ 1,  6, 11, 16],  
       [ 2, 14, 12, 17],  
       [ 3,  8, 13, 18],  
       [ 4,  9, 14, 19]])
```

C. MASKI LOGICZNE

Generuję nowe przykładowe dane zaczynające się od -10 a kończące na 10 z krokiem 0.5

```
[ ] A = np.arange(start=-10, stop=10, step=0.5)
A
```

```
array([-10. , -9.5, -9. , -8.5, -8. , -7.5, -7. ,
        -5.5, -5. , -4.5, -4. , -3.5, -3. , -2.5,
        -1. , -0.5, 0. , 0.5, 1. , 1.5, 2. ,
        3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5,
        8. , 8.5, 9. , 9.5])
```

Zmieniam kształt tablicy na 10 WIERSZY

```
[ ] A = A.reshape(10, -1)
A
```

```
array([[ -10. ,  -9.5,  -9. ,  -8.5],
       [  -8. ,  -7.5,  -7. ,  -6.5],
       [  -6. ,  -5.5,  -5. ,  -4.5],
       [  -4. ,  -3.5,  -3. ,  -2.5],
       [  -2. ,  -1.5,  -1. ,  -0.5],
       [   0. ,   0.5,   1. ,   1.5],
       [   2. ,   2.5,   3. ,   3.5],
       [   4. ,   4.5,   5. ,   5.5],
       [   6. ,   6.5,   7. ,   7.5],
       [   8. ,   8.5,   9. ,   9.5]])
```

Chcę się dowiedzieć w których miejscach mam wartości ujemne. Zwraca false gdy te wartości są mniejsze od 0

```
[ ] A > 0
```

```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

Przekazuję maskę do tablicy i wycinam tylko dodatnie elementy

```
[ ] A[A > 0]
```

```
array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5,
       7. , 7.5, 8. , 8.5, 9. , 9.5])
```

Łączę dwa warunki

```
[ ] np.bitwise_and(A > -5, A < 5)
```

```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

```
[ ] np.bitwise_or(A < -5, A > 5)
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

```
[ ] A[np.bitwise_or(A < -5, A > 5)]
```

```
array([-10. , -9.5, -9. , -8.5, -8. , -7.5, -7. , -6.5, -6. ,
        -5.5,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,
         9.5])
```

FUNKCJA PONIŻSZA - zwraca maskę tj. prawdę i fałsz, dzięki której wycinam dane i otrzymuje z przedziału od -5 do 5

```
[ ] A[np.bitwise_and(A > -5, A < 5)]
```

```
array([-4.5, -4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5,  0. ,  0.5,
        1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
```

KONIEC

